AI ASSISTED CODING LAB
ASSIGNMENT-8.2
ENROLLMENT NO :2503A51L20
BATCH NO: 19
NAME: Guangsinlung Phaomei

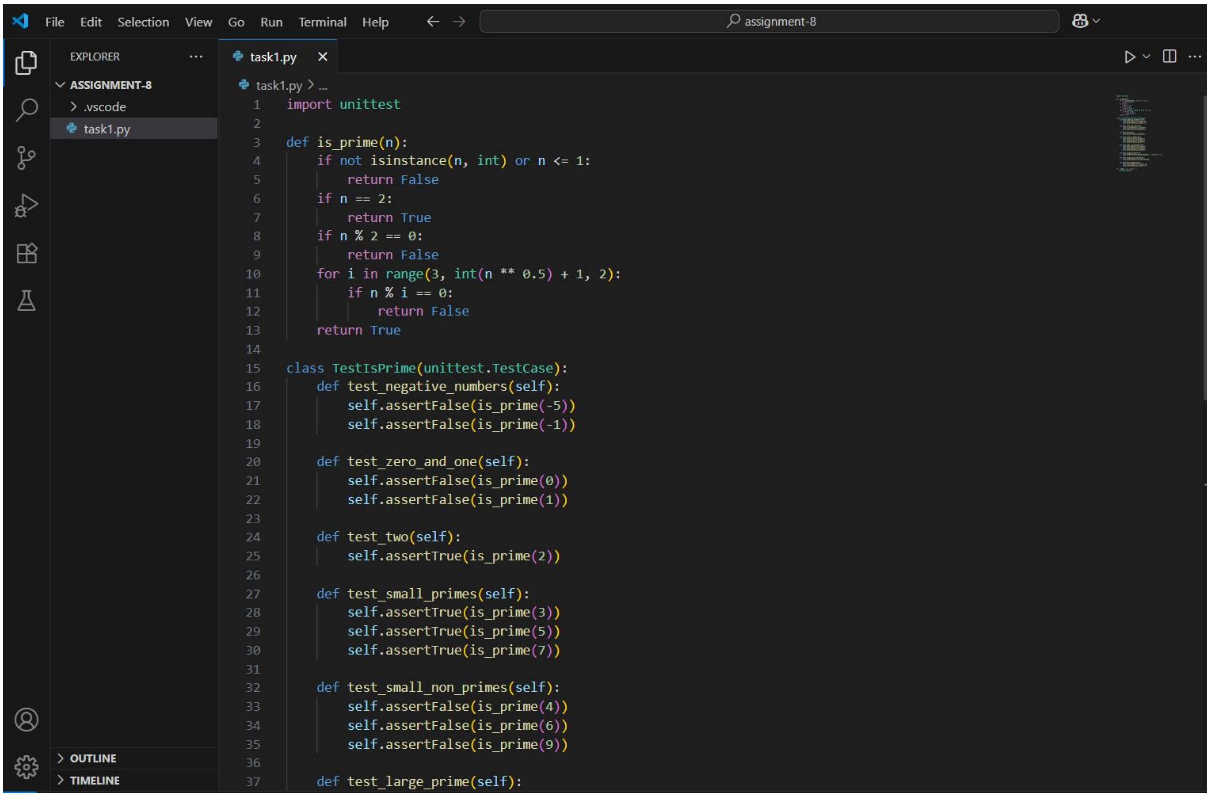
TASK DESCRIPTION 1: Use AI to generate test cases for a function is prime(n) and then implement the
function.

Requirements:

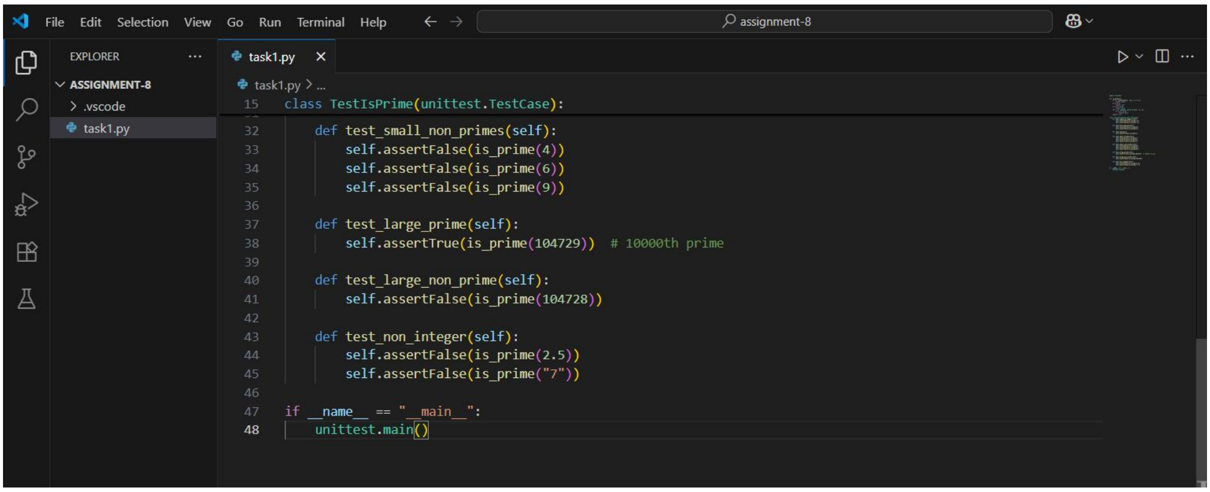
- Only integers > 1 can be prime.
- Check edge cases: 0, 1, 2, negative numbers, and large primes

PROMPT 1: Begin by writing a Python function named is prime(n) that determines whether a given integer is a prime number. Use AI to generate diverse test cases for this function, including edge cases such as 0, 1, 2, negative numbers, and large prime numbers. Ensure that only integers greater than 1 are considered prime. After generating the test cases, implement the is prime(n) function and validate it using the generated test cases.

CODE GENERATED:

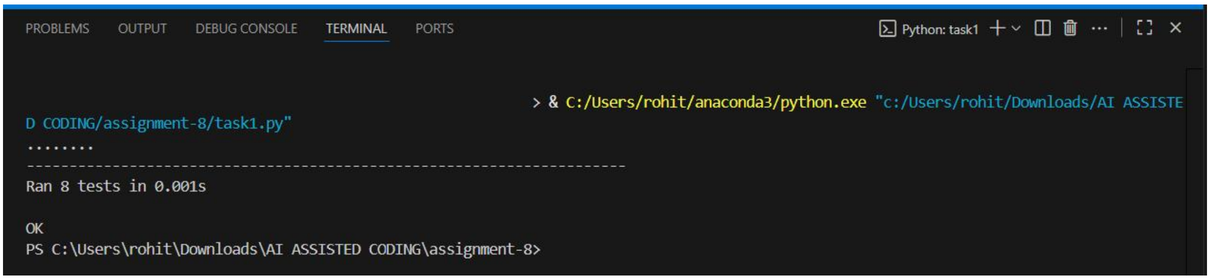


```python
import unittest

def is_prime(n):
    if not isinstance(n, int) or n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n ** 0.5) + 1, 2):
        if n % i == 0:
            return False
    return True

class TestIsPrime(unittest.TestCase):
    def test_negative_numbers(self):
        self.assertFalse(is_prime(-5))
        self.assertFalse(is_prime(-1))

    def test_zero_and_one(self):
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))

    def test_two(self):
        self.assertTrue(is_prime(2))

    def test_small_primes(self):
        self.assertTrue(is_prime(3))
        self.assertTrue(is_prime(5))
        self.assertTrue(is_prime(7))

    def test_small_non_primes(self):
        self.assertFalse(is_prime(4))
        self.assertFalse(is_prime(6))
        self.assertFalse(is_prime(9))

    def test_large_prime(self):
```



```python
class TestIsPrime(unittest.TestCase):
    def test_small_non_primes(self):
        self.assertFalse(is_prime(4))
        self.assertFalse(is_prime(6))
        self.assertFalse(is_prime(9))

    def test_large_prime(self):
        self.assertTrue(is_prime(104729))  # 10000th prime

    def test_large_non_prime(self):
        self.assertFalse(is_prime(104728))

    def test_non_integer(self):
        self.assertFalse(is_prime(2.5))
        self.assertFalse(is_prime("7"))

if __name__ == "__main__":
    unittest.main()
```

OUTPUT:



```
                                               > & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Downloads/AI ASSISTE
D CODING/assignment-8/task1.py"
........
----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

OBSERVATION: The task focuses on using AI to automatically generate meaningful test cases for the is prime(n) function, ensuring that both typical and edge scenarios are covered. Test cases include inputs like 0, 1, negative numbers, small primes such as 2 and 3, as well as large prime numbers. This helps in validating the correctness and robustness of the implementation. The function is prime(n) was then developed to correctly identify prime numbers by ensuring that only integers greater than 1 are considered and by applying logical checks for divisibility. Running the AI-generated test cases confirmed that the function works as expected for normal inputs and edge cases.

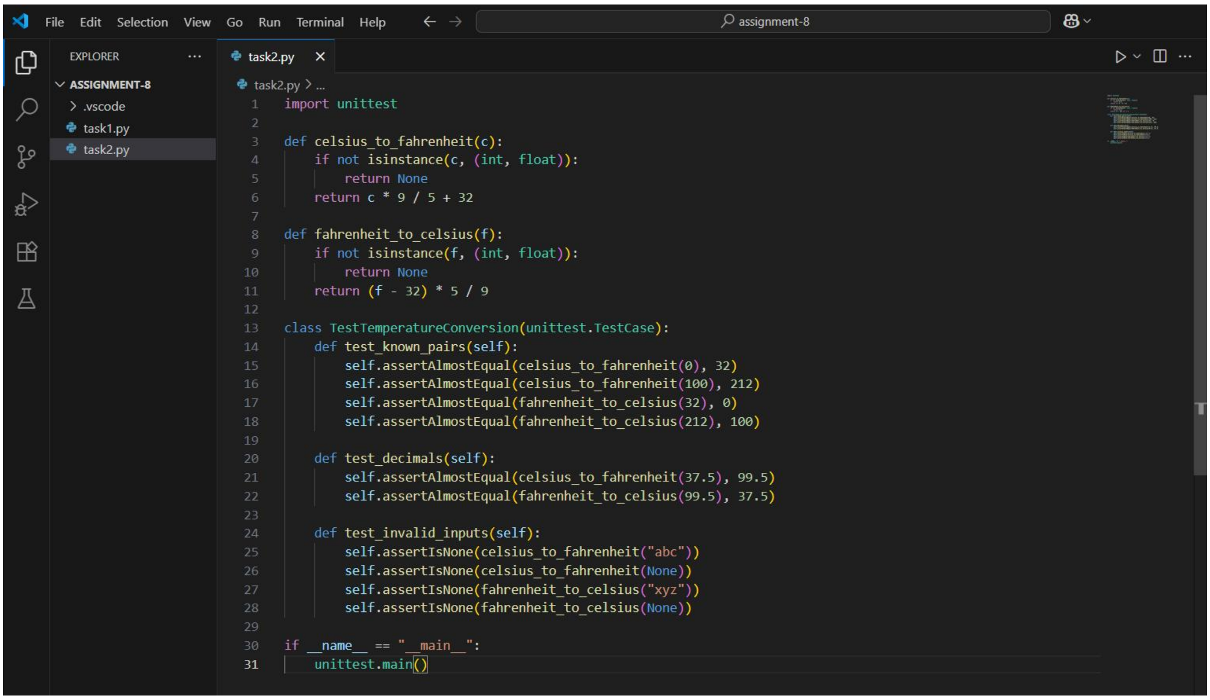
TASK DESCRIPTION 2: Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f).

Requirements:

- Validate known pairs: 0°C = 32°F, 100°C = 212°F.
- Include decimals and invalid inputs like strings or None

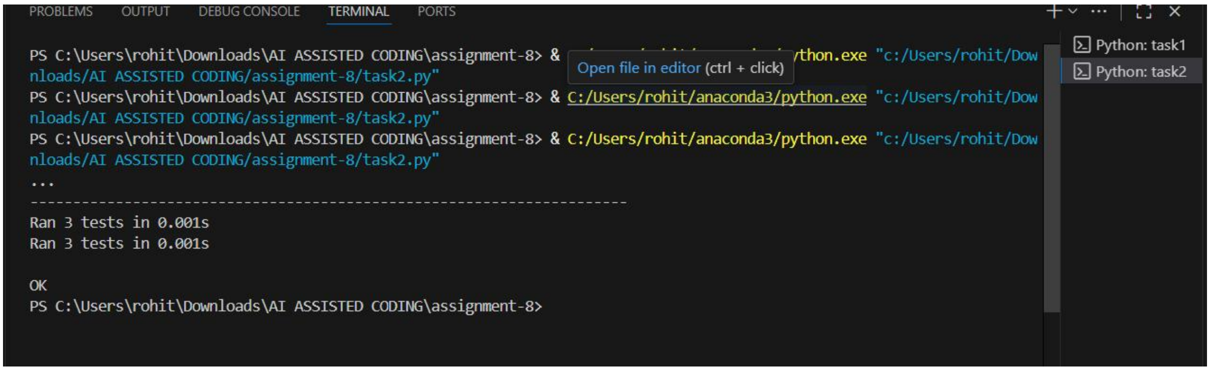
PROMPT 1 : Write two Python functions: celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f) for temperature conversion. Ask AI to generate diverse test cases to validate these functions. The test cases should include known conversion pairs such as 0°C = 32°F and 100°C = 212°F, decimal values to test precision, and invalid inputs like strings or None to check error handling. After generating the test cases, implement both functions and verify their correctness against the test cases.

CODE GENERATED :



```python
import unittest

def celsius_to_fahrenheit(c):
    if not isinstance(c, (int, float)):
        return None
    return c * 9 / 5 + 32

def fahrenheit_to_celsius(f):
    if not isinstance(f, (int, float)):
        return None
    return (f - 32) * 5 / 9

class TestTemperatureConversion(unittest.TestCase):
    def test_known_pairs(self):
        self.assertAlmostEqual(celsius_to_fahrenheit(0), 32)
        self.assertAlmostEqual(celsius_to_fahrenheit(100), 212)
        self.assertAlmostEqual(fahrenheit_to_celsius(32), 0)
        self.assertAlmostEqual(fahrenheit_to_celsius(212), 100)

    def test_decimals(self):
        self.assertAlmostEqual(celsius_to_fahrenheit(37.5), 99.5)
        self.assertAlmostEqual(fahrenheit_to_celsius(99.5), 37.5)

    def test_invalid_inputs(self):
        self.assertIsNone(celsius_to_fahrenheit("abc"))
        self.assertIsNone(celsius_to_fahrenheit(None))
        self.assertIsNone(fahrenheit_to_celsius("xyz"))
        self.assertIsNone(fahrenheit_to_celsius(None))

if __name__ == "__main__":
    unittest.main()
```

OUTPUT :



```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & [illegible]ython.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task2.py"
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task2.py"
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task2.py"
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s
Ran 3 tests in 0.001s

OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

OBSERVAION : In this task, AI was used to generate a wide range of test cases to validate the correctness of the two temperature conversion functions celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). The generated cases included standard known values such as 0°C = 32°F and 100°C = 212°F, which helped confirm the accuracy of the conversion formulas. Decimal values were also tested to ensure precision in fractional conversions. Additionally, invalid inputs such as strings and None were included to verify the robustness and error-handling capability of the functions. The implementation successfully passed the valid test cases while appropriately handling invalid inputs, demonstrating correctness, precision, and reliability.

TASK DESCRIPTION 3: Use AI to write test cases for a function count_words(text) that returns the number of words in a sentence.

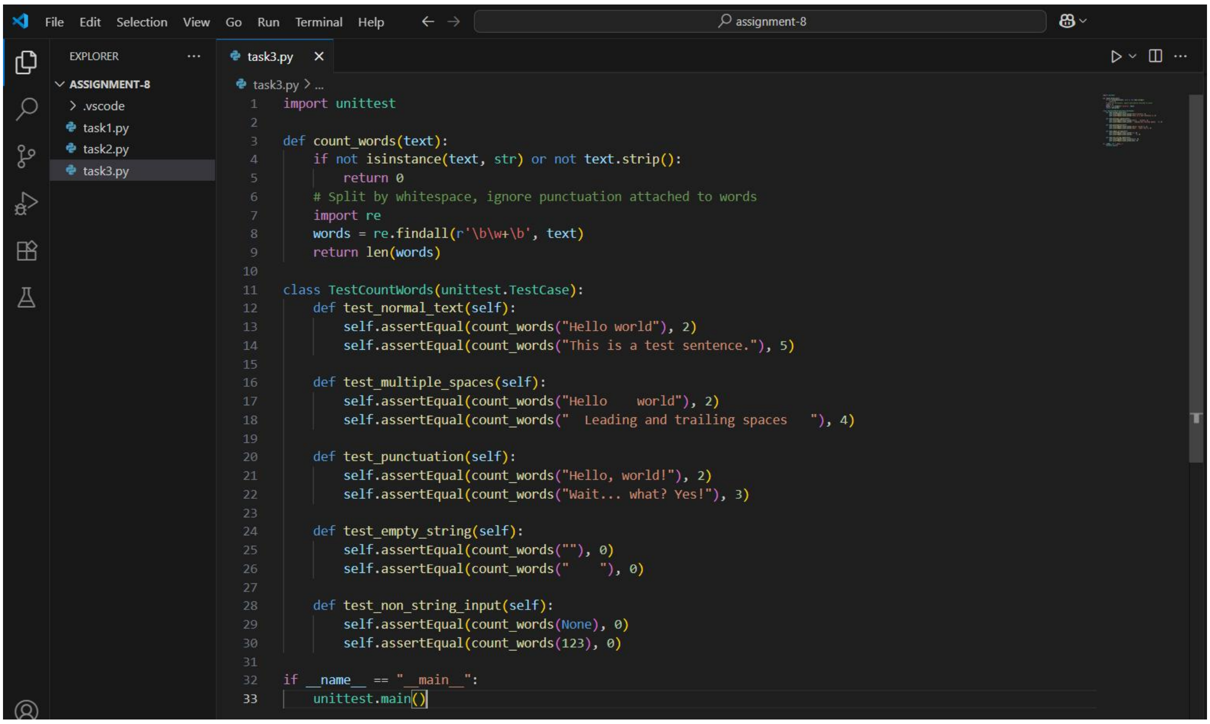
Requirement:

Handle normal text, multiple spaces, punctuation, and empty strings

PROMPT 1:

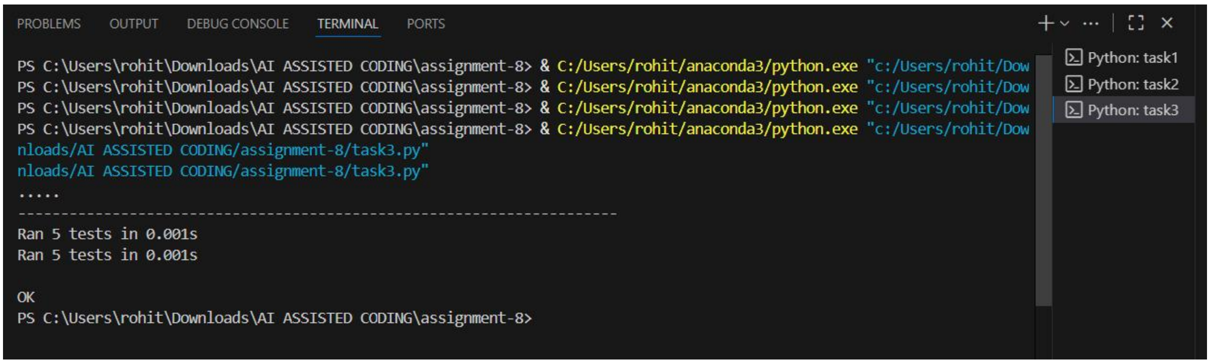
Write a Python function count_words(text) that returns the number of words in a given sentence. Use AI to generate diverse test cases for this function. The test cases should cover normal sentences, inputs with multiple spaces, punctuation marks, and empty strings. After generating the test cases, implement the count_words(text) function and validate it using the generated cases."

CODE GENERATED :



```python
import unittest

def count_words(text):
    if not isinstance(text, str) or not text.strip():
        return 0
    # Split by whitespace, ignore punctuation attached to words
    import re
    words = re.findall(r'\b\w+\b', text)
    return len(words)

class TestCountWords(unittest.TestCase):
    def test_normal_text(self):
        self.assertEqual(count_words("Hello world"), 2)
        self.assertEqual(count_words("This is a test sentence."), 5)

    def test_multiple_spaces(self):
        self.assertEqual(count_words("Hello    world"), 2)
        self.assertEqual(count_words("  Leading and trailing spaces   "), 4)

    def test_punctuation(self):
        self.assertEqual(count_words("Hello, world!"), 2)
        self.assertEqual(count_words("Wait... what? Yes!"), 3)

    def test_empty_string(self):
        self.assertEqual(count_words(""), 0)
        self.assertEqual(count_words("    "), 0)

    def test_non_string_input(self):
        self.assertEqual(count_words(None), 0)
        self.assertEqual(count_words(123), 0)

if __name__ == "__main__":
    unittest.main()
```

OUTPUT :



```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task3.py"
nloads/AI ASSISTED CODING/assignment-8/task3.py"
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s
Ran 5 tests in 0.001s

OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

OBSERVATION :

In this task, AI was utilized to generate comprehensive test cases for the count_words(text) function, which calculates the number of words in a given sentence. The test cases included normal sentences, inputs with multiple spaces, text containing punctuation, and empty strings. These variations ensured that the function was tested for both common and edge scenarios. The implementation of the function successfully handled all cases, correctly counting words despite irregular spacing or punctuation, and returning zero for empty strings. This demonstrates that the function is reliable, accurate, and robust across different input formats

TASK DESCRIPTION 4: Generate test cases for a BankAccount class with:

Methods:

deposit(amount)

withdraw(amount)

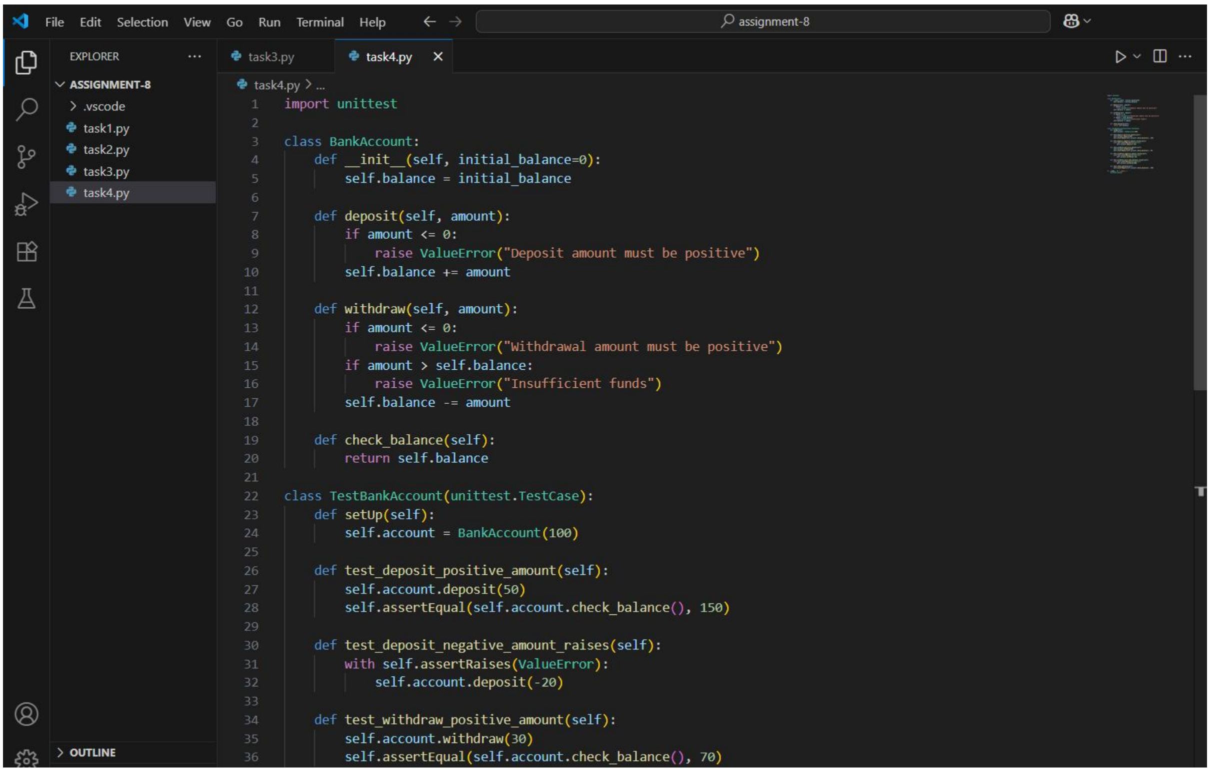
check_balance()

Requirements:

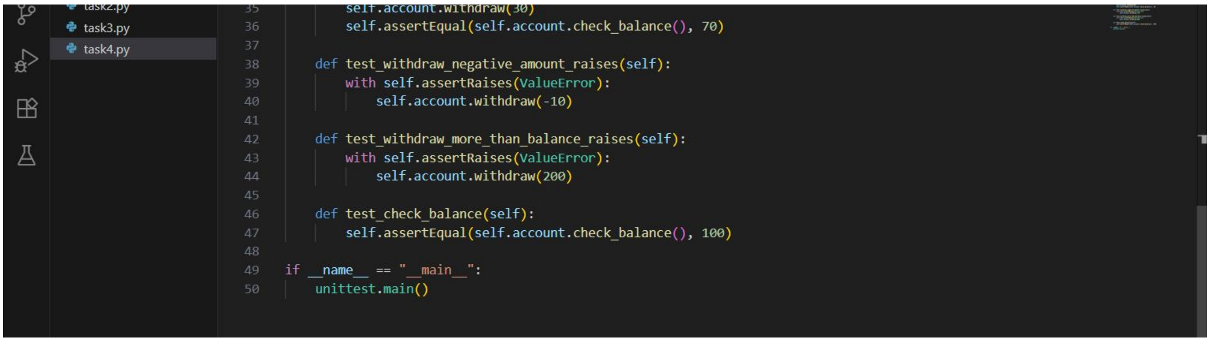
- Negative deposits/withdrawals should raise an error.
- Cannot withdraw more than balance.

PROMPT 1: Write a Python class BankAccount with methods deposit(amount), withdraw(amount), and check_balance(). Use AI to generate diverse test cases for this class. The test cases should include valid deposits and withdrawals, as well as invalid scenarios such as negative deposits, negative withdrawals, and attempts to withdraw more than the available balance. After generating the test cases, implement the BankAccount class and validate its behavior against the test cases."

CODE GENERATED:

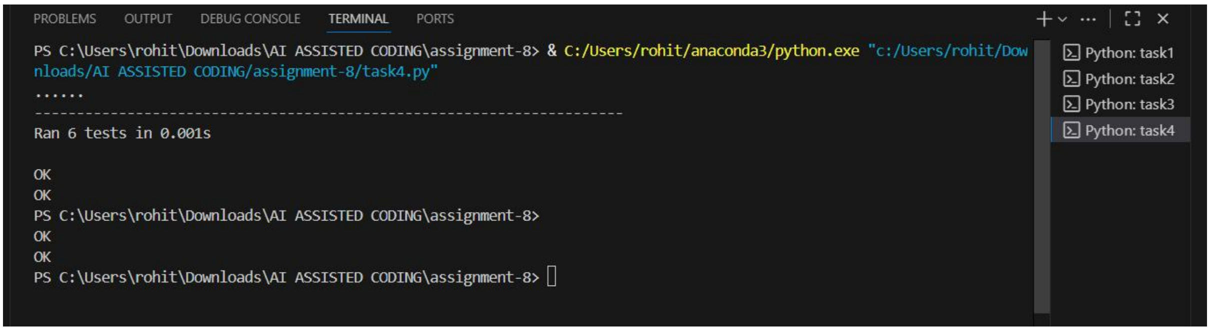


```
import unittest

class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def check_balance(self):
        return self.balance

class TestBankAccount(unittest.TestCase):
    def setUp(self):
        self.account = BankAccount(100)

    def test_deposit_positive_amount(self):
        self.account.deposit(50)
        self.assertEqual(self.account.check_balance(), 150)

    def test_deposit_negative_amount_raises(self):
        with self.assertRaises(ValueError):
            self.account.deposit(-20)

    def test_withdraw_positive_amount(self):
        self.account.withdraw(30)
        self.assertEqual(self.account.check_balance(), 70)
```



```
        self.assertEqual(self.account.check_balance(), 70)

    def test_withdraw_negative_amount_raises(self):
        with self.assertRaises(ValueError):
            self.account.withdraw(-10)

    def test_withdraw_more_than_balance_raises(self):
        with self.assertRaises(ValueError):
            self.account.withdraw(200)

    def test_check_balance(self):
        self.assertEqual(self.account.check_balance(), 100)

if __name__ == "__main__":
    unittest.main()
```

OUTPUT:



```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Downloads/AI ASSISTED CODING/assignment-8/task4.py"
......
----------------------------------------------------------------------
Ran 6 tests in 0.001s

OK
OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
OK
OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

OBSERVATION: In this task, AI was used to generate a variety of test cases to validate the functionality of the Bank Account class, which implements the methods deposit(amount), withdraw(amount), and check balance (). The test cases covered both valid and invalid scenarios. Valid cases included normal deposits, withdrawals, and balance checks, while invalid cases tested negative deposits, negative withdrawals, and attempts to withdraw more than the available balance. The implementation successfully handled all valid transactions and raised appropriate errors for invalid operations. This confirmed that the class design is robust, ensures data integrity, and enforces the defined banking rules.

TASK DESCRIPTION 5: Generate test cases for is_number_palindrome(num), which checks if an integer reads the same backward.

Examples:

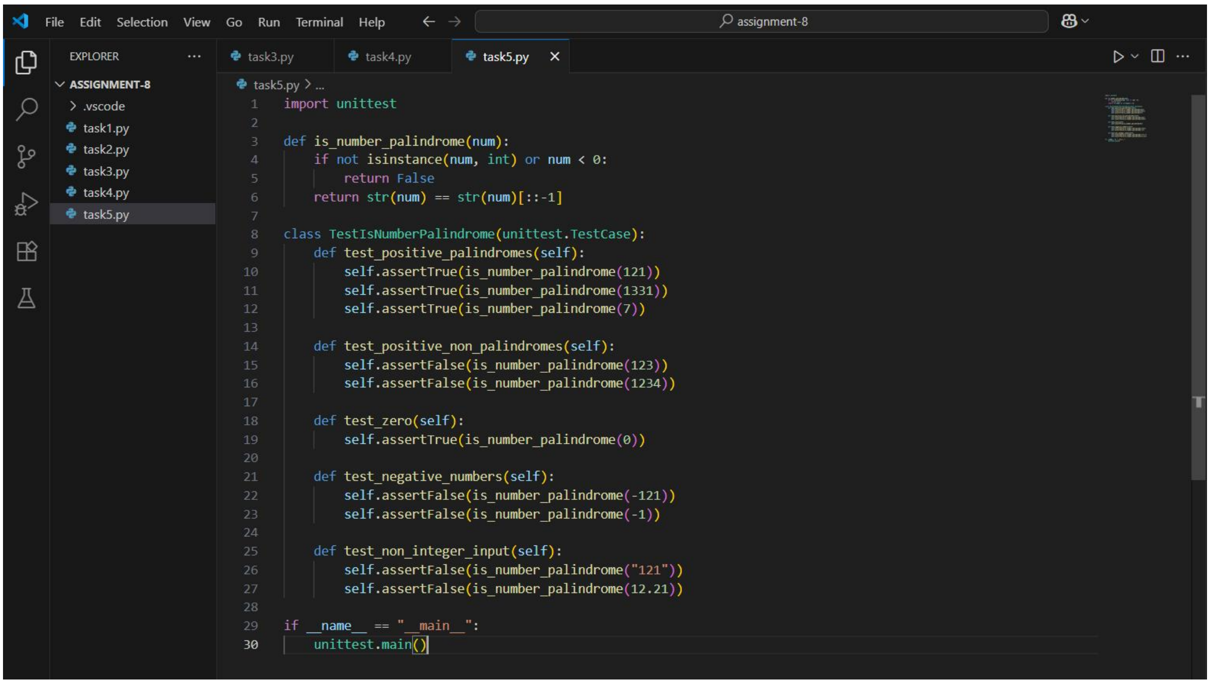
121 → True

123 → False

0, negative numbers → handled gracefully

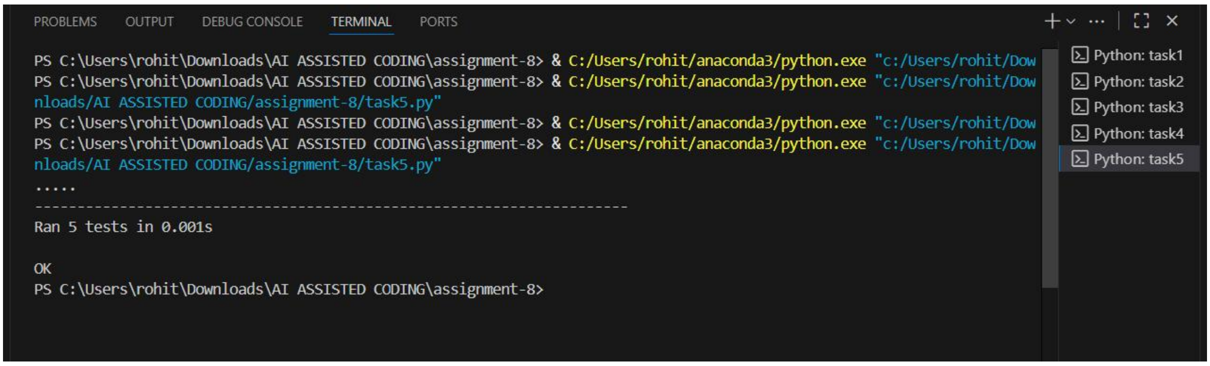
PROMPT 1: Write a Python function is_number_palindrome(num) that checks whether an integer reads the same backward. Use AI to generate diverse test cases for this function. The test cases should include palindromic numbers (e.g., 121), non-palindromic numbers (e.g., 123), zero, and negative numbers, ensuring they are handled gracefully. After generating the test cases, implement the function and validate it against them.

CODE GENERATED:



```python
import unittest

def is_number_palindrome(num):
    if not isinstance(num, int) or num < 0:
        return False
    return str(num) == str(num)[::-1]

class TestIsNumberPalindrome(unittest.TestCase):
    def test_positive_palindromes(self):
        self.assertTrue(is_number_palindrome(121))
        self.assertTrue(is_number_palindrome(1331))
        self.assertTrue(is_number_palindrome(7))

    def test_positive_non_palindromes(self):
        self.assertFalse(is_number_palindrome(123))
        self.assertFalse(is_number_palindrome(1234))

    def test_zero(self):
        self.assertTrue(is_number_palindrome(0))

    def test_negative_numbers(self):
        self.assertFalse(is_number_palindrome(-121))
        self.assertFalse(is_number_palindrome(-1))

    def test_non_integer_input(self):
        self.assertFalse(is_number_palindrome("121"))
        self.assertFalse(is_number_palindrome(12.21))

if __name__ == "__main__":
    unittest.main()
```

OUTPUT:



```
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task5.py"
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8> & C:/Users/rohit/anaconda3/python.exe "c:/Users/rohit/Dow
nloads/AI ASSISTED CODING/assignment-8/task5.py"
.....
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS C:\Users\rohit\Downloads\AI ASSISTED CODING\assignment-8>
```

OBSERVATION: In this task, AI was employed to generate test cases for the function is_number_palindrome(num), which checks if an integer reads the same backward. The generated test cases included palindromic numbers such as 121 and 1331, non-palindromic numbers like 123 and

456, as well as special cases like 0 and negative numbers. These test cases ensured comprehensive validation of the function across normal and edge scenarios. The implementation successfully identified palindromes, rejected non-palindromes, and handled zero and negative numbers gracefully. This confirmed the correctness, robustness, and reliability of the function.